Extended Essay in Computer Science

Investigating the performance of the Rabin-Karp Algorithm and the Knuth-Morris-Pratt algorithm

How does the performance of the Knuth-Morris-Pratt algorithm compare to the Rabin-Karp Algorithm for finding patterns in textual data as the text and pattern sizes increase?

Session: May 2022

Essay word count: 3823 Words

Table of contents

1. Introduction

As the world becomes increasingly dependent on the use of documents and databases, string matching — the process of finding the occurrences of a group of characters within a large text — becomes very crucial. This field of computer science can be applied in numerous scenarios: algorithms for detecting plagiarism, spell checkers; and even plays a very fundamental role in search engines which involves matching strings through a large database (Soni et al. 3:6372). Even in the medical field, computer science can be applied to find patterns within a given genetic sequence.

The most fundamental method of string matching, known as the brute force method, involves matching the text with the pattern, character by character, which is very inefficient. Since then, numerous algorithms have been developed which are based on the original one and utilizes very different methods of searching and matching.

Due to the constant development, there exist a vast number of pattern searching algorithms—and a range of applications—which means that the performance and suitability for a given situation (i.e text size, occurrences of pattern, text format) could vary. This establishes a diverse group of algorithms, each equipped with their strengths and weaknesses affecting their suitability for any given task involving string matching.

This essay will be exploring two different pattern searching algorithms, Knuth-Morris-Pratt and Rabin-Karp, which are to be tested against a variety of text samples. The aim of the experimentation and research is to gather insight as to whether KMP is a better and more effective string matching algorithm than

Rabin-Karp for finding patterns in a given textual data—taking increasing text and pattern sizes into consideration.

Computer science is an area of study that not only involves the continuous development of algorithms to find a more efficient and faster process, it is also about the correct application of those algorithms to solve problems in the real world. The research question is worthy of an investigation to find and compare the two algorithms to gather data and assess their effectiveness — in terms of speed and performance — for their uses. Furthermore, gathering data will reveal characteristics of the algorithms that could act as useful information during the decision making of which string matching algorithm is suitable for a given task.

2. Theoretical Background

2.1 Time Complexity

Time complexity refers to the total time taken for a specific program to run, this is calculated by taking into account the steps required until the completion of the program ("Time Complexity"). For example, a program that has 2 iterations of ***n*** times will have a time complexity of n^2. Due to variations of outcomes that can occur, the time complexity usually showcases the running time for the worst-case scenario which is the maximum that is required to perform a certain algorithm. **Figure 2.1.1** shows some examples of time complexity in which there are different relations between the running time of a program with the input size.

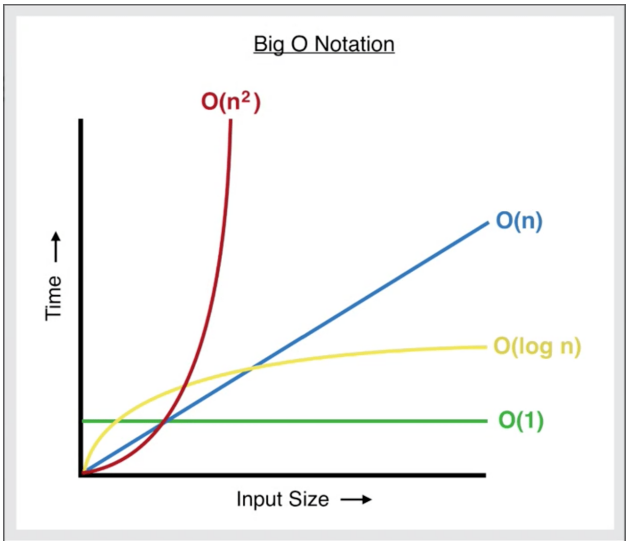


Figure 2.1.1: Different variations of time complexities (Kaiser)

Understanding the concept is crucial for the comparison of effectiveness between two different algorithms. The metric shows the weakness and strengths of the algorithm and how the input size affects the performance of the program. The essay will be referring to the time complexity to summarize how different input sizes affect the performance.

Another metric of performance is space complexity, however, it will not be explored in depth as storage space tends to be reusable and the algorithms discussed use very minimal extra space making the discussion irrelevant to the overall performance.

2.2 String Matching Algorithms

In summary, string matching algorithms find the number of occurrences a certain pattern can be found within a given textual data. In most of the algorithms, this is done by comparing a chunk of the text with the characters in the pattern. Texts can be treated as an array of characters and assuming the size of the pattern is ***m***, we can start from the left side of the array containing the actual text, a method of comparison depending on the algorithm is performed within the specific 'window' of size ***m*** (Cormen 988). Every time the checking and matching are performed on the text, a shift to the right of the text occurs. However, if the character to be checked does not match with the first character of the pattern, no further checking will be performed as the algorithm can deduce that it is a mismatch.

If, according to the algorithm, a match is found between the pattern and a specific section of the text, the algorithm will output the index where the pattern is found. **Figure 2.2.1** depicts a visual representation of the naive algorithm where the pattern can be found at indexes: 3, 7,9, and 13.

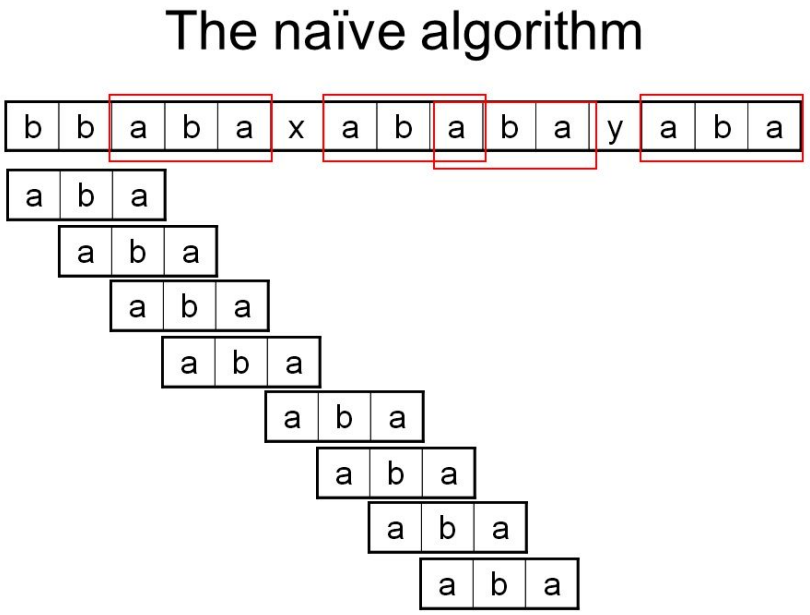


Figure 2.2.1: General Process of String Matching algorithms (Stephens)

This is the most fundamental process and method regarding string matching which is often known as the *Naive Algorithm*. Algorithms like KMP and Rabin-Karp which are to be investigated in this paper will be based on this concept, however, they are implemented with different methods. As both are very complex algorithms, understanding this core concept will be essential for exploration.

2.3 KMP Algorithm

Although a naive algorithm is easy to implement, there are flaws when compared to other methods of string matching. Assuming that we have a text that consists of ***n*** characters and a pattern of characters of size ***m***, there would be ***n-m+1*** possible shifts that can be performed. For each of the possible shifts mentioned, ***m*** number of comparisons would have to be done for the characters to conclude whether the pattern and the string match for a specific window. As a result, the running time for the worst-case scenario would be **O((n-m+1)*m)** (Cormen 988). **Figure 2.2.1** shows that for a naive algorithm, a nested loop is needed as every shift requires a comparison of ***m*** times (the length of pattern).

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4       if P[1 . . m] == T[s + 1 . . s + m]
5            print "Pattern occurs with shift" s
```

Figure 2.3.1: Pseudocode for Naive algorithm (Cormen 988)

The KMP algorithm solves the issue where the naive algorithm doesn't perform any analysis on the pattern and text; Even though we already know that a section of text already matches the pattern, the original algorithm continues to perform the matching which increases the execution time. With KMP, whenever a mismatch occurs, an algorithm is performed that predetermines whether some initial characters are the same. When this happens, some of the characters are skipped which is more efficient and saves more time compared to the naive algorithm. The KMP algorithm makes use of the concept of prefix and suffix to identify the longest substring that appears more than once within a pattern.

Using that information, the algorithm predetermines the substrings that are repeated more than once within a pattern using the concept of prefix and suffix — which can be skipped whenever a shift occurs after a mismatch. In **Figure 2.3.2**, as str1 is the same as str2, after the mismatch, the algorithm skips str1 in the next shift as we already know that it will match str3.

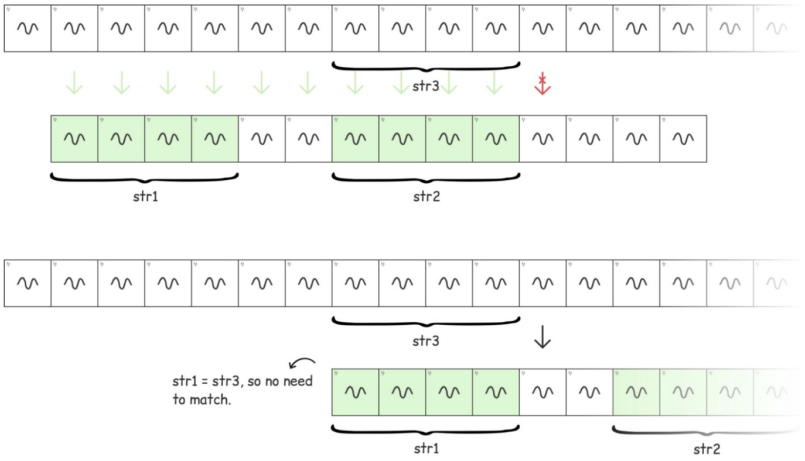


Figure 2.3.2: Diagram depicting the concept of prefix and suffix (Budhwani)

The array that contains the information for the occurrences of prefix and suffix is denoted as π[i], where ***i*** is the size of the pattern. Tables similar to **Figure 2.3.4** are useful as whenever a mismatch occurs at, for example, S[2] or S[5]-S[8], we can use π[i] to decide which comparison can be skipped, thus saving more computation time. Because the text that is being compared contains a substring that occurs more than once, whenever a mismatch is to occur at S[9] the algorithm will utilize the pi table and start at S[5] during the next shift as we know that “aabc” is repeated twice in the pattern.

i	1	2	3	4	5	6	7	8	9	10
$s[i]$	a	a	b	c	a	a	b	c	d	a
$\pi(i)$	0	1	0	0	1	2	3	4	0	1

Figure 2.3.4: pi table (Pöial)

We can represent the concept of the prefixes and suffixes mathematically as shown in **Figure 2.3.5** where ***s*** is the longest prefix that is also a suffix and it should not be equal to the total pattern size of ***i***.

$$\pi[i] = \max_{k=0\ldots i} \{k : s[0 \ldots k-1] = s[i-(k-1) \ldots i]\}$$

Figure 2.3.5: Mathematical representation of prefix and suffix ("Prefix Function")

The time complexity for this algorithm is ***O(m+n)*** as the full algorithm consists of a preprocessing phase of ***O(m)*** and a comparison phase of ***O(n)*** where ***m*** is the size of the pattern and ***n*** is the size of the text being searched and compared (Crochemore and Lecroq 9). Due to the need to determine prefixes and suffixes, the algorithm makes use of an array and has a space complexity of ***O(m)***, where ***m*** is the size of the pattern (Laud).

2.4 Rabin Karp

An alternative version of the naive algorithm implements the concept of hashing tables. This avoids the need for an algorithm that compares characters one by one. To do this, the Rabin-Karp algorithm computes the total hash value for a specific pattern and compares it with the total value for the specific window of the text. Although the nature of the algorithm is quite different, the comparison still shifts and cycles through the textual data. See **Figure 2.4.1** for a visual representation of the process.

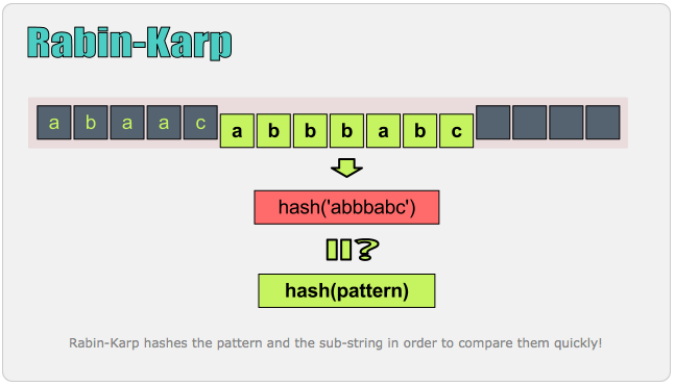


Figure 2.4.1: General process for Rabin-Karp (Popov)

Rabin-Karp utilizes a hash function that takes in a series of characters and maps them to a specific numerical value. Essentially, it allows the letters within a given string or text to be converted into numerical values with the help of a hash table (Techopedia). Assuming the letters “A” to “D” are assigned a value from 1 to 5. For a given text “ABCD”, the total value would be **H(A)+H(B)+H(C)+H(D)**, the result would be 10. However, the downside of such a concept is that a collision could occur, this is when the hash value generated from different inputs is the same which is considered an error, sometimes also known as a spurious hit. For example, the texts “ABCD” and “DCBA” generate the same hash value despite being different texts which could

be a problem as the algorithm could mistake these values to match with the pattern despite the difference in the structure of the text. This issue can be minimized by multiplying each letter of the textual data by a constant to the power of the position of the letters, with this method the position of the letters will impact the total hash value generated. As a result, windows of text with the same letters in different positions can lead to the computation of different hash values. For a pattern of size ***m***, the computation of the total hash value from a section of a given text would use an algorithm from **Figure 2.4.2**.

$$H = c_1 \times b^{m-1} + c_2 \times b^{m-2} + c_3 \times b^{m-3} ... + c_m \times b^0$$

c = characters in the string **m** = length of string **b** = constant

Figure 2.4.2: Hash Algorithm (Block)

One of the main requirements for the computation of a hash function is that it has to be efficiently computable using the current hash value and the value of the next character to perform a shift of window (Geekforgeeks). The Rabin-Karp algorithm efficiently calculates the hash value for the next window by including the hash value of the current window. The algorithm used is called the rolling hash and it can be represented by **Figure 2.4.3**.

$$H = (H_p - C_p \times b^{m-1}) \times b + C_n$$

Hp = previous hash **Cp** = previous character **Cn** = new character **m** = window size **b** = constant

Figure 2.4.3: Rolling Hash Algorithm (Block)

The hash value of the first letter is calculated and subtracted from the total and the value for the next letter is added to the total. However, as the number of characters and patterns increases, the size of the constant multiplied by the position of the characters increases, this would result in large hash values, thus, a modulo would be applied. Creating an improved algorithm represented as **H(total)** = H % Q where Q is usually a large prime number.

In terms of time complexity, the Rabin-Karp algorithm has an average case of **O(m+m)** whereas the rolling hash algorithm takes **O(n)** to complete (where ***n*** is the total number of characters to be checked and ***m*** is the number of characters in the pattern). To make sure that the pattern matches the section of the textual data being compared, ***m*** number of comparisons have to be made. As a result, the worst-case time complexity would be **O(mn)**, this is true for cases where every single hash comparison results in the same value as the hash value for the pattern (Popov; Geekforgeeks). To make sure that the pattern matches with the specific window of the text, ***m*** number of comparisons have to be made. Rabin-Karp has a space complexity of O(1) meaning that no space is required to search the pattern (Kumar).

3. Methodology

3.1 Experimental Methodology

The experiment will consist of two sections, one which varies the size of the text against a pattern that consists of 5 characters/digits and the second varies the pattern size against a fixed textual data.

1. The textual data and pattern for each experiment will be randomly generated using specific programs and algorithms
2. Data will be passed into the respective programs (refer to *appendix* section A for the programs that will be used to find the patterns given the textual data)
3. The time taken to execute the programs/operation will be outputted and inserted into a spreadsheet
4. The average time taken to execute the programs will be calculated for each textual data and their category

3.2 Data sets used to conduct the experiment

The steps described will be repeated for different types of experiments where the textual data will vary. This is to ensure that the functionality of the experiments are covered holistically and the two algorithms are compared to decide which algorithm is the most suitable for the respective dataset. For each data type, the number of characters in the textual data will be changing to decide whether the input size affects the computational time for each algorithm and how they compare against each other.

Lists of data types to be investigated and rationale for the choices:

- Numbers: generated randomly where the purpose is to find out how effective the algorithms are in terms of finding patterns related to numbers
- DNA sequences: also generated randomly which consists of only 4 letters that have a real-life application
- Dummy text: computer-generated texts that appear to look like a normal text
- Randomly generated strings

3.3 Dependent Variables

The variables that will be changing within the experiment will be the time taken to execute the pattern searching by Rabin Karp and KMP algorithms. As computation can fluctuate, any noticeable outliers are removed from the data set when compared to other repetitions of the same set to minimize the chances of inaccurate results

3.4 Hardware and Software

To minimize the number of random and systematic errors which can alter the final result, certain variables are kept consistent whether it is hardware or software, see **Figure 3.4.1** for the table of all the devices and software used to experiment.

Type	Description
Hardware	Device: Macbook Pro (Early 2015) Processor: 2,7 GHz Dual-Core Intel Core i5 Memory: 8 GB 1867 MHz DDR3
Software	Operating System: macOS Catalina, version 10.15.7 Platform: Python (Visual Studio Code)

Figure 3.4.1: Table of software and hardware

Due to the nature of the programs, the programming language used to execute the algorithms would not affect the result of the experiment. The choice of Python as the programming language is purely based on its ease of usability, such as built-in functions like timeit where the execution time for a snippet of code can be given. Built-in features like these make python the perfect choice for measuring the performance of the program, in terms of time and speed.

4 Results

4.1 Tabular and Graphical presentation

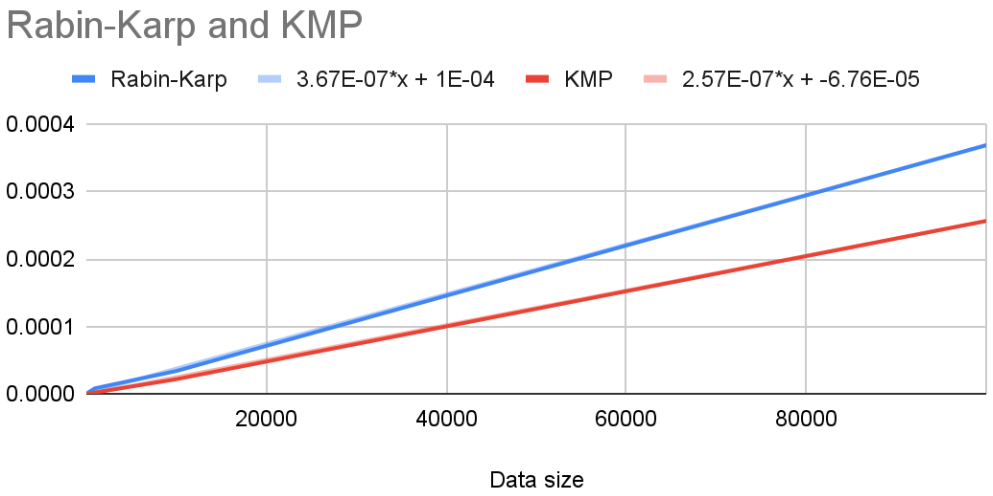
The experiments were repeated with different data types, each repeated 5 times and the average value is calculated. Refer to the *appendix section B* for the full result and the sample data types used to calculate the results.

4.1.1 Variation in text size:

5 digit numbers within texts of numbers (size) 10-100000

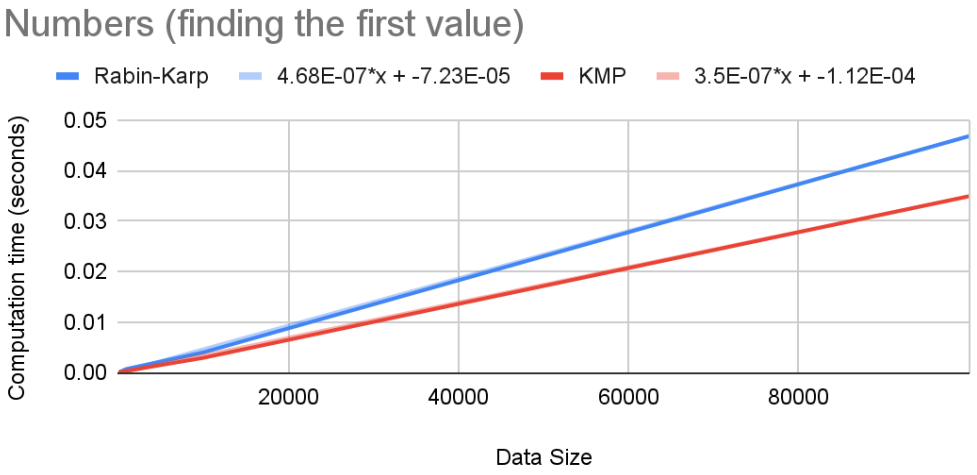
Data size	Rabin-Karp	KMP
10	0.000042582	0.000036194
100	0.000120404	0.000057412
1000	0.000870608	0.000252868
10000	0.003416822	0.00220876
100000	0.036864044	0.02564931

Table 1: Average execution time to find pattern within a large string of numbers



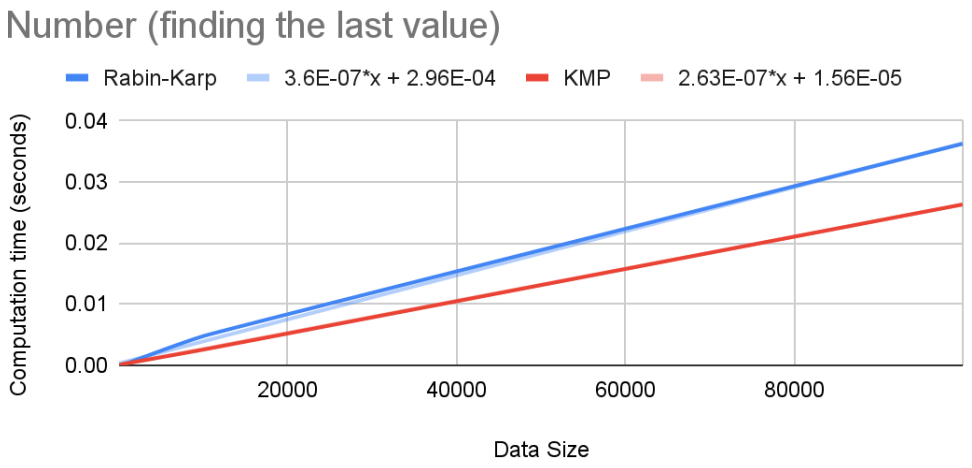
Data size	Rabin-Karp	KMP
10	0.000049638	0.000043248
100	0.000070382	0.000070524
1000	0.000733806	0.000352048
10000	0.004001188	0.002926398
100000	0.04680538	0.034911824

Table 2: Average execution time to find pattern at the first index within a large string of numbers



Data size	Rabin-Karp	KMP
10	0.000099184	0.00005045
100	0.000069572	0.000057794
1000	0.000337412	0.000319148
10000	0.004762364	0.002545404
100000	0.036241148	0.026304818

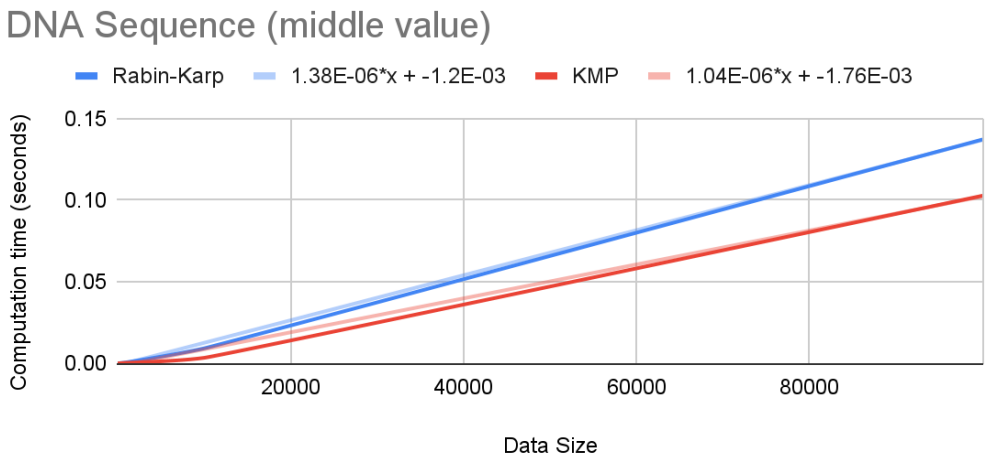
Table 3: Average execution time to find pattern at the last index within a large string of numbers



Finding DNA Sequence within text of characters (size) 10-100000

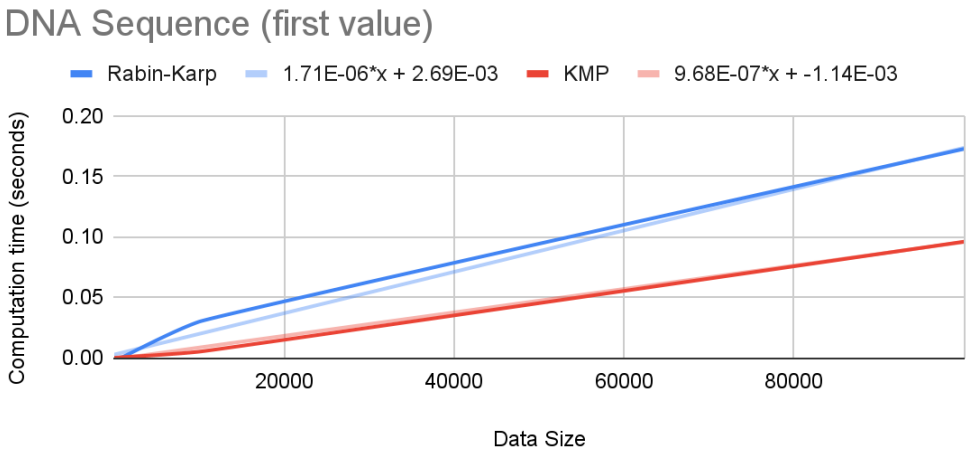
Data size	Rabin-Karp	KMP
10	0.000171758	0.000083064
100	0.00012479	0.00007963
1000	0.000660754	0.000418138
10000	0.009240244	0.003442432
100000	0.137176372	0.102749394

Table 4: average execution time to find a dna sequence within a large text of string consisting of (a,g,c,and t)



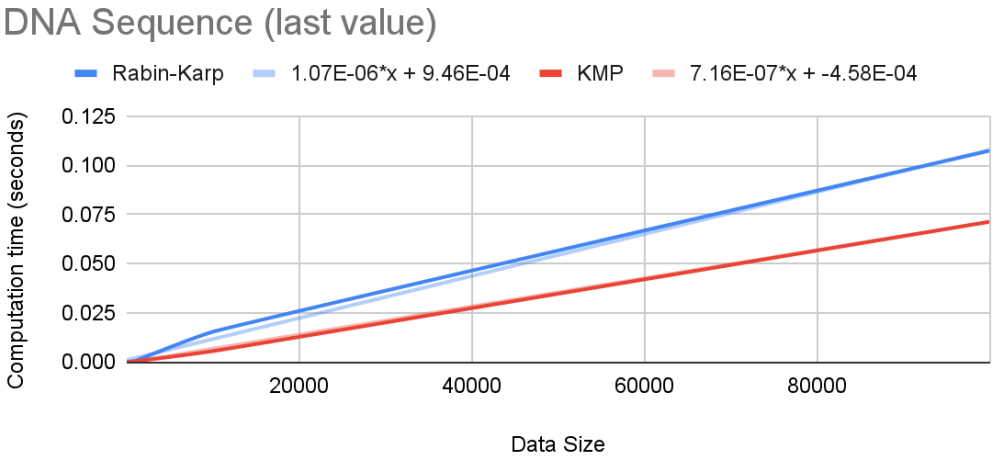
Data size	Rabin-Karp	KMP
10	0.00006466	0.000045346
100	0.000119066	0.000092794
1000	0.000693418	0.000629804
10000	0.029844712	0.005108786
100000	0.172662592	0.09603653

Table 5:average execution time to find a dna sequence at the first index within a large text of string consisting of (a,g,c,and t)



Data size	Rabin-Karp	KMP
10	0.000069808	0.000053214
100	0.000083732	0.00007467
1000	0.000515222	0.000360682
10000	0.015335416	0.005520964
100000	0.107398604	0.071304082

Table 6: average execution time to find a dna sequence at the last index within a large text of string consisting of (a,g,c,and t)

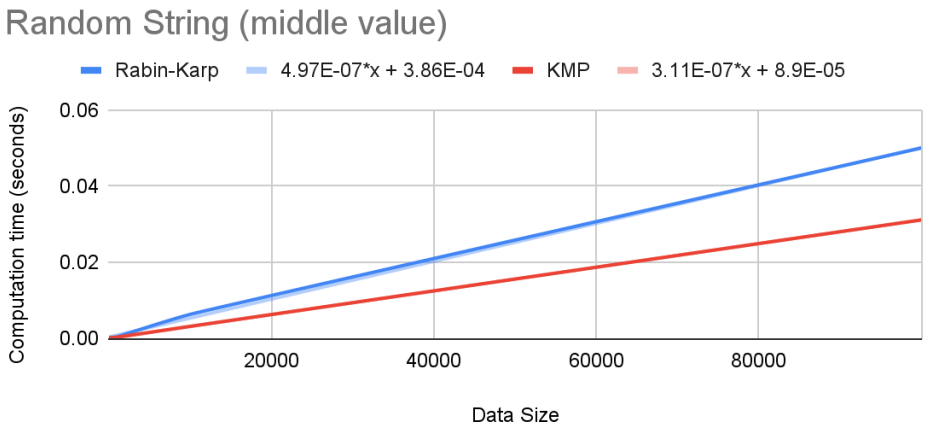


Sequence of characters within text of randomly generated string (size)

10-100000

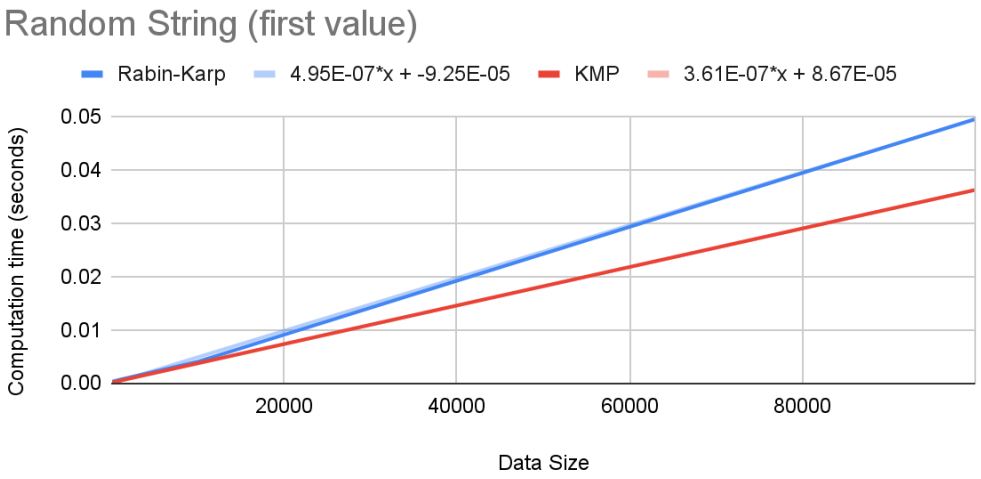
Data size	Rabin-Karp	KMP
10	0.000099708	0.000097178
100	0.000277044	0.000193596
1000	0.000450276	0.00033102
10000	0.00633502	0.003182982
100000	0.050028898	0.03115039

Table 7: The average execution time to find a sequence of characters within a large randomly generated string (ranging from a-z)



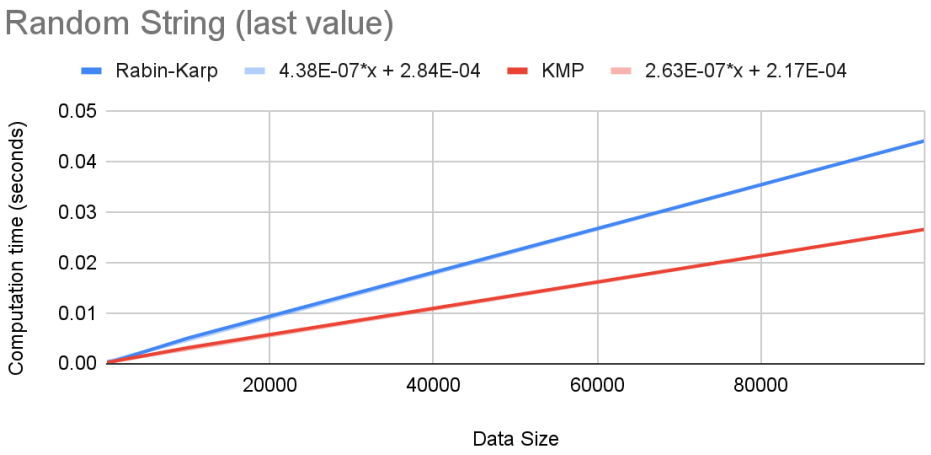
Data size	Rabin-Karp	KMP
10	0.00006957	0.000071812
100	0.000296212	0.000102806
1000	0.000660992	0.000467636
10000	0.00402422	0.003721236
100000	0.049509002	0.036218406

Table 9: Average execution time to find a sequence of characters within a large randomly generated string at the first index (ranging from a-z)



Data size	Rabin-Karp	KMP
10	0.000238036	0.000177004
100	0.000275374	0.000078724
1000	0.00048628	0.00044594
10000	0.005038214	0.003120328
100000	0.044059802	0.026527834

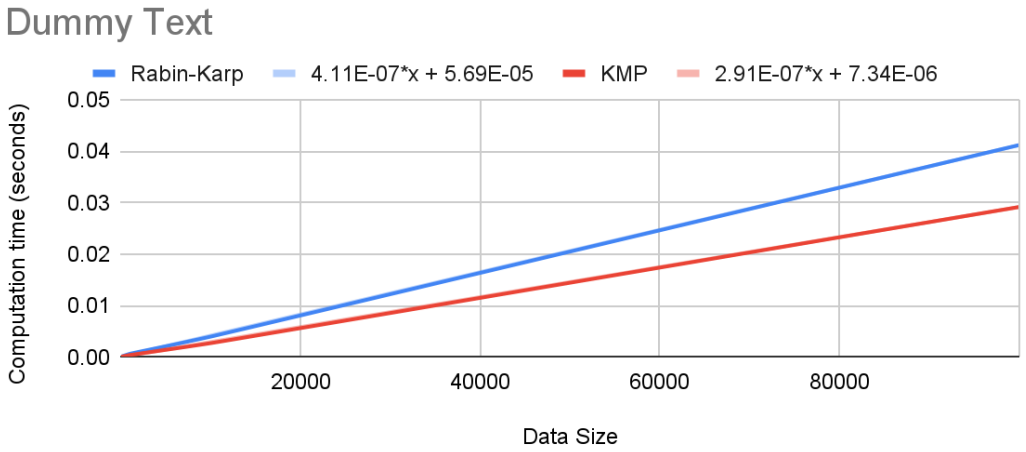
Table 10: Average execution time to find a sequence of characters within a large randomly generated string at the last index (ranging from a-z)



Words within a dummy text (size) 10-100000

Data size	Rabin-Karp	KMP
10	0.000044776	0.000049068
100	0.000124454	0.000096416
1000	0.000638342	0.00039916
10000	0.003971002	0.002700998
100000	0.041199158	0.029166222

Table 10: average execution time to find a word from a randomly generated dummy text

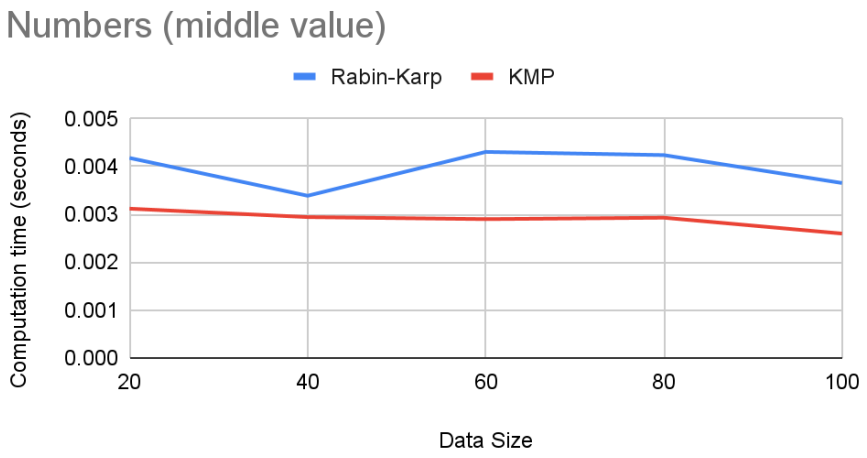


4.1.2 Variation in pattern sizes:

10-50 digit numbers within texts of numbers size 10000

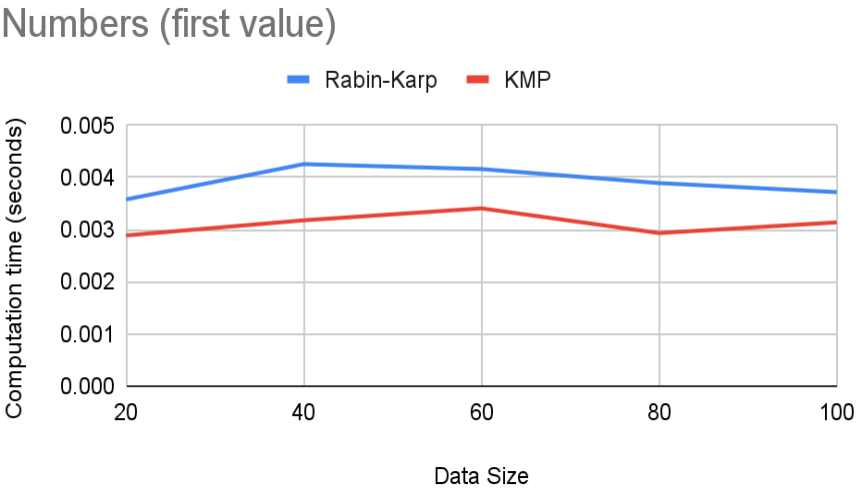
Data size	Rabin-Karp	KMP
20	0.004179764	0.00311885
40	0.003389168	0.002942658
60	0.004306222	0.002900076
80	0.004239466	0.002929974
100	0.003655674	0.002598192

Table 11: average execution time to find numbers of increasing digits within a set sequence of numbers



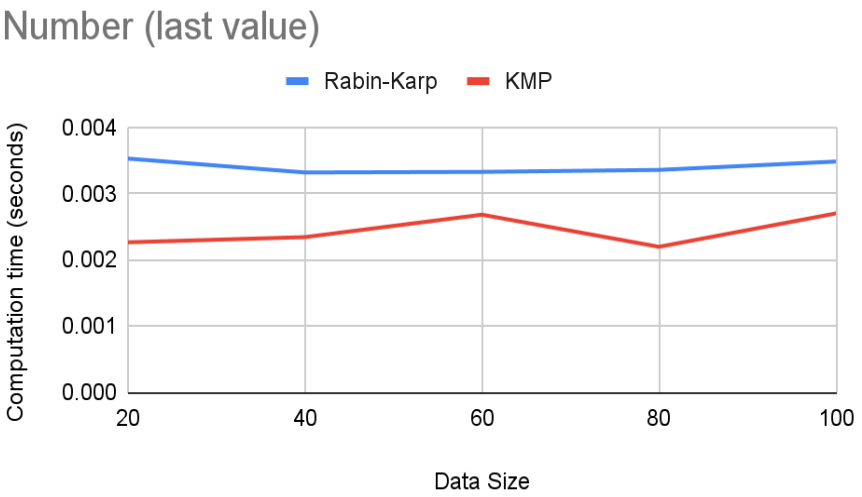
Data size	Rabin-Karp	KMP
20	0.003579762	0.002888584
40	0.004257392	0.00317898
60	0.004162408	0.003408192
80	0.00389347	0.002933932
100	0.003719664	0.003139164

Table 12: average execution time to find numbers of increasing digits at the first index within a set sequence of numbers



Data size	Rabin-Karp	KMP
20	0.00352745	0.002264642
40	0.003317786	0.002343038
60	0.00332651	0.002681398
80	0.003356408	0.002197028
100	0.00348258	0.00270152

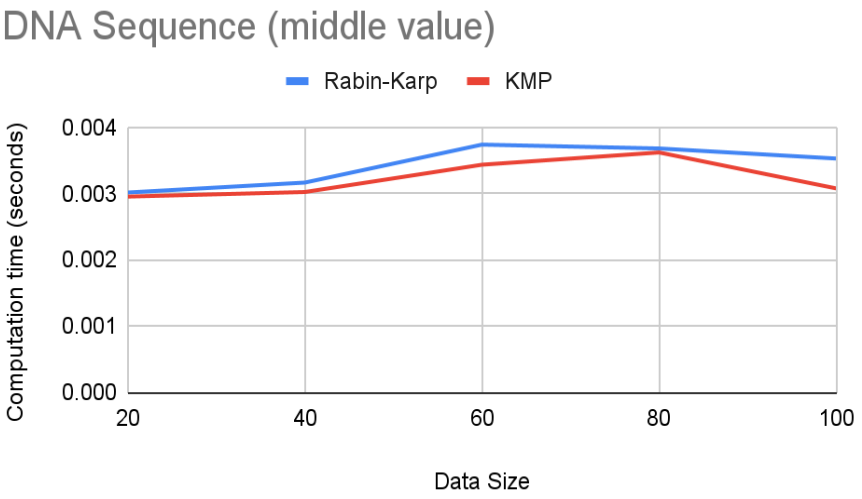
Table 13: average execution time to find numbers of increasing digits at the last index within a set sequence of numbers



10-50 characters of DNA sequence within texts of size 10000

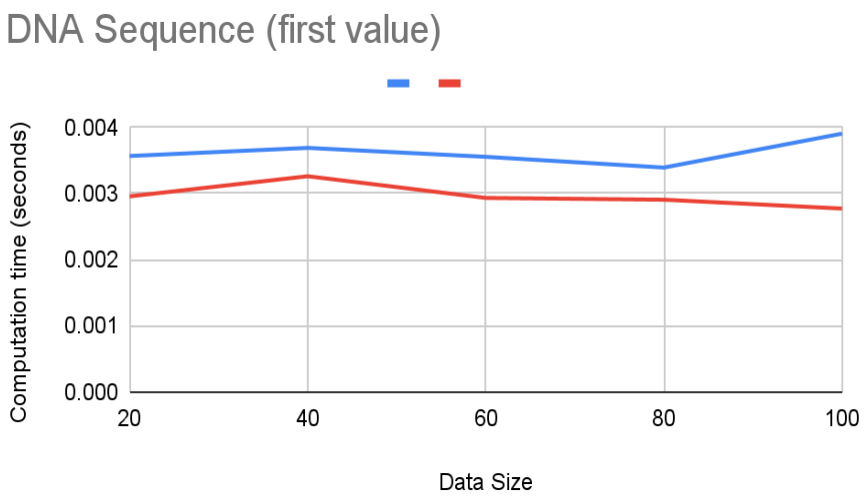
Data size	Rabin-Karp	KMP
20	0.003015374	0.002953766
40	0.003165388	0.003023004
60	0.003737782	0.003434658
80	0.003679798	0.003619624
100	0.003527976	0.00307765

Table 14: average execution time to find dna sequences of increasing size within a set sequence of string consisting of (a, g, c, and t)



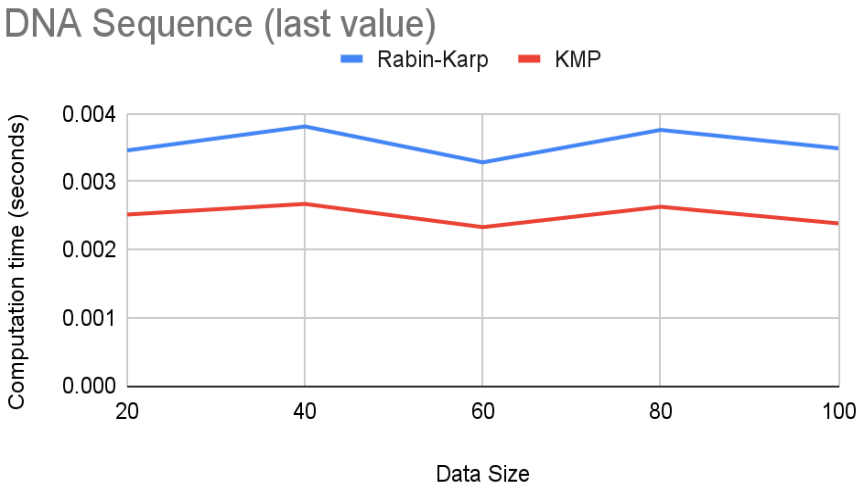
Data size	Rabin-Karp	KMP
20	0.00356793	0.00295763
40	0.00369043	0.003261852
60	0.003555394	0.00293374
80	0.00339279	0.002906988
100	0.003905774	0.00277295

Table 15: average execution time to find dna sequences of increasing size at the first index within a set sequence of string consisting of (a, g, c, and t)



Data size	Rabin-Karp	KMP
20	0.003459646	0.002517698
40	0.003812264	0.00267396
60	0.003284598	0.00233283
80	0.00376148	0.002632046
100	0.0034904	0.002387522

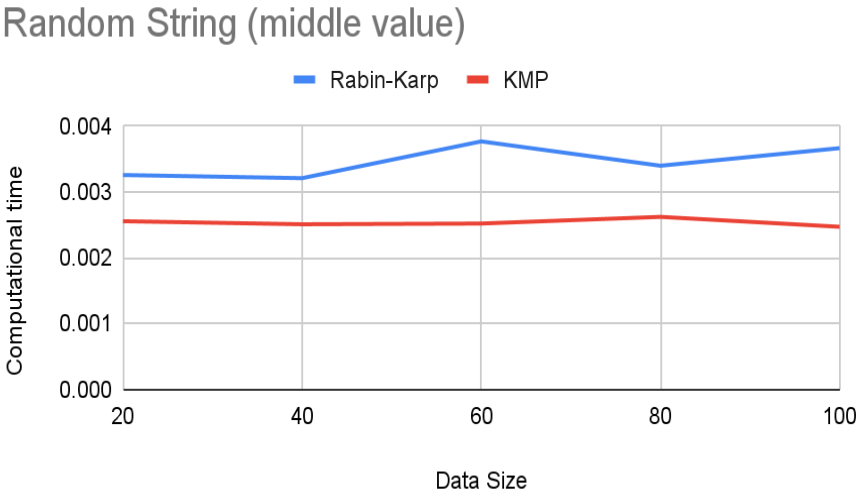
Table 16: average execution time to find dna sequences of increasing size at the last index within a set sequence of string consisting of (a, g, c, and t)



10-50 characters within a randomly generated text of size 10000

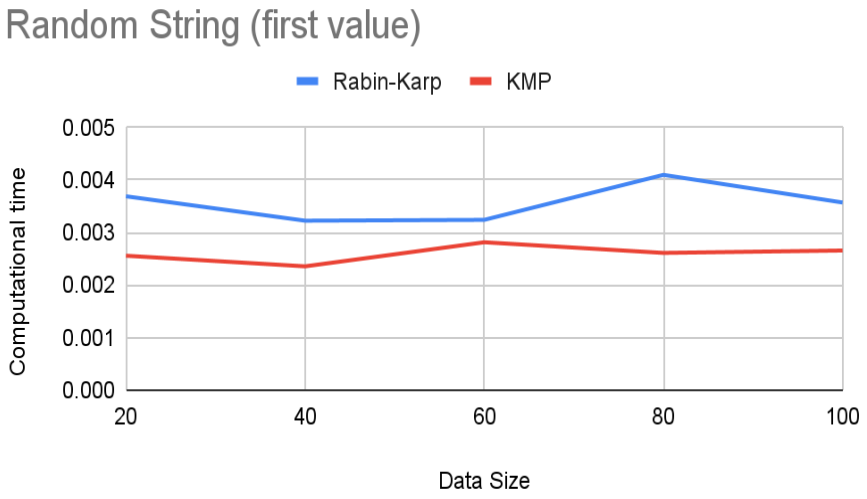
Data size	Rabin-Karp	KMP
20	0.003257128	0.00255561
40	0.003209354	0.002509164
60	0.003767586	0.00252037
80	0.003398036	0.002621554
100	0.003665254	0.00247097

Table 17: average execution time to find characters of increasing size within a randomly generated string of text



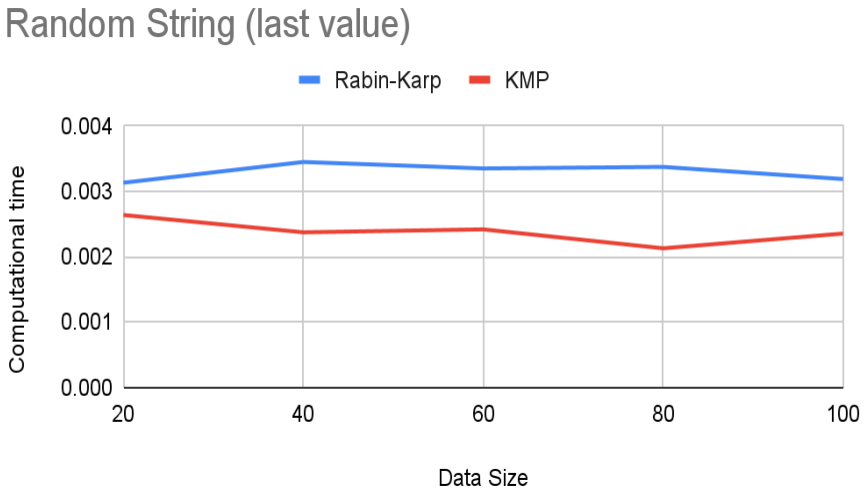
Data size	Rabin-Karp	KMP
20	0.003690196	0.00255771
40	0.003225182	0.002355622
60	0.003241108	0.002814054
80	0.004098226	0.002611446
100	0.003571366	0.002657272

Table 18: average execution time to find characters of increasing size at the first index within a randomly generated string of text



Data size	Rabin-Karp	KMP
20	0.00313177	0.002638148
40	0.00344901	0.00237317
60	0.003350212	0.002418424
80	0.003373622	0.00212884
100	0.00318794	0.0023541

Table 19: average execution time to find characters of increasing size at the last index within a randomly generated string of text



4.2 Data Analysis

General results show that the KMP algorithm performed significantly better than Rabin-Karp in all the categories. It is noticeable that as the text size increases, the margin between the two algorithms also increases, indicating that Rabin-Karp doesn't perform as well against large textual data. A possible reason is that Rabin-Karp deals with large hash values and a lot of false matches due to the large data size which increases computation time. All the graphs in the first section of the experiment showed a linear relationship between the text size and the computation time. This is generally shown when the input size is increased by 10 fold, the computation time is also multiplied by 10, thus, establishing a constant increase. However, Rabin-Karp generally has a steeper gradient in terms of the trendline which shows that sizes of texts have more effects on the performance. The experiment also showed that positions of the pattern have very little effect on the computation time of both algorithms as the results don't display any significant trend.

To compare the two algorithms, the area of margins between the graphs can be calculated by integrating the equations of the trendline.

$$\int_{10}^{100000} R(x) - K(x)\,dx$$

Where R(x) is the equation of the trendline for Rabin-Karp and K(x) is the equation of the trendline for the KMP algorithm

Figure 4.2.1 General Process of Integration

The results of the calculations are as following:

Table number	Area of Margin (unit²)	Average
Table 1	553	550
Table 2	586	
Table 3	513	
Table 4	1644	3468
Table 5	3891	
Table 6	4869	
Table 7	960	837
Table 8	671	
Table 9	882	
Table 10	605	605

Table 20: Area of Margin between KMP and Rabin-Karp using integration

The first section of the experiment where the algorithms have to find a pattern of numbers shows the smallest margin. Numbers have smaller ASCII values which significantly reduces the total hash values for Rabin-Karp, hence, a reduced time of completion.

The second experiment, involving DNA sequences, shows a large margin between the performance of the two algorithms. One of the possible explanations is that DNA matching takes up more computation time for both algorithms, thus, while dealing with a larger magnitude of numbers, the margin itself increases. However, upon further analysis of the sample data and results, the reason points to the characteristics of the algorithms. The structure of the DNA sequence samples only consists of four letters: A, G, C, and T. Even with a good hashing function, the chances of producing the same hash value for different character arrangements increases. As mentioned, the Rabin-Karp algorithm has a worst-case scenario when every single comparison results in a collision and further comparison needs to be conducted to ensure that the characters match. With such a limited pool of character arrangements, the occurrences of false hash matching increase which is very inefficient and unproductive as it only lengthens the computation time.

The result for the experiment involving dummy text is surprising as Rabin-Karp has a range of applications like plagiarism checkers, yet, the performance time of KMP is still superior. The popularity of Rabin-Karp may be due to its simplicity and flexibility, but in theory, the KMP is more suitable for dummy text.

Meanwhile, experimental results from **table 7-9** show that Rabin-Karp performs weaker with randomly generated strings of characters. After analysing the dataset passed into the algorithm, randomly generated strings can include letters with high hashing values like “x”,”y”, and “z” which increases the amount of calculation needed. On the other hand, this should not have affected KMP as the comparison is

direct, rather than, numerical which meant that the increase in computational time is only influenced by the increase in data size.

As for the second part of the experiment which deals with the increase in pattern sizes, unlike the first part, the computation time shows minor fluctuations and no obvious increase in time taken to execute the programs. To show a clear relationship between the two variables, the pattern sizes would have to range into thousands in values which is difficult to conduct. On average, the computation time across the last 9 experiments are very similar, which can be concluded that pattern sizes don't affect computation time unless done on a scale that shows significant differences.

4.3 Limitations

Due to limited resources, many limitations were presented during the experimentation stage.

- In the second part of the experiment, incrementing the size of the pattern is difficult to achieve, as a result, the experimental results only showed minor fluctuations.
- It is very difficult to keep the occurrences of a particular pattern in terms of finding DNA sequences due to limited combinations of letters.
- Hardware components used to experiment could also affect the computation time which could lead to slight inaccuracies in the results.

5. Conclusions

Referring to the question, "*How does the performance of the Knuth-Morris-Pratt algorithm compare to the Rabin-Karp Algorithm for finding patterns in textual data as the text and pattern sizes increase?*" From my research, Rabin-Karp is favourable for real-life applications as it supports multi-pattern searching. However, the experiment showcased that KMP is, in most cases, the fastest algorithm in terms of time and speed to find patterns of textual data with increasing sizes. Although Rabin-Karp performed worse, it doesn't necessarily conclude that it is completely replaceable by the KMP algorithm. With a space complexity of O(1), the algorithm is very cost-efficient — especially when dealing with large amounts of data. It is also much more flexible in terms of customization as there are many variations of Rabin-Karp with different hash value calculations and methods of shifting characters. As a result, it tends to be the favourable string matching algorithm as it can be altered to be used for plagiarism checkers. However, KMP is much more reliable as it guarantees that the results are 100% correct.

The experiment also showed that the performance of Rabin-Karp is directly dependent on the hash values associated with the letters/strings which is not favourable when dealing with large texts. Meanwhile, the computation speed of KMP is affected by the pattern size, data size and the structure of the pattern (whether it includes prefixes and suffixes). This makes KMP a better performing algorithm for matching textual data—especially for large data size as it is influenced by fewer variables.

With more resources available, the research question can be expanded by changing more conditions, for example, the number of pattern occurrences or experimentation involving a larger number of text type variations to fully evaluate the effectiveness of each algorithm which is crucial as efficiency and reliability are important factors to consider for companies or medical facilities that tackles strings in daily operations, especially on a larger scale.

Bibliography

"Applications of String Matching Algorithms." GeeksforGeeks, 3 Sept. 2020, www.geeksforgeeks.org/applications-of-string-matching-algorithms/. Accessed 11 June 2021.

Block, Vigar. "Rabin-Karp Algorithm Using Polynomial Hashing and Modular Arithmetic." Medium, 6 Oct. 2020, medium.com/swlh/rabin-karp-algorithm-using-polynomial-hashing-and-modular-arithmetic-437627b37db6. Accessed 30 Sept. 2021.

Kaiser, Colton. “Big o Time Complexity: What It Is and Why It Matters for Your Code.” *Medium*, Level Up Coding, 27 May 2020, https://levelup.gitconnected.com/big-o-time-complexity-what-it-is-and-why-it-matters-for-your-code-6c08dd97ad59. Accessed 11 June 2021.

Cormen, Thomas H, and Thomas H. Cormen. *Introduction to Algorithms*. Cambridge, Mass: MIT Press, 2001. Print.

Crochemore, Maxime, and Thierry Lecroq. "Pattern-Matching and Text-Compression Algorithms." ACM Comput. Surv, vol. 28, 1 Mar. 1996, pp. 7-9.

"KMP Algorithm for Pattern Searching." GeeksforGeeks, 24 Mar. 2021, www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/. Accessed 7 Apr. 2021.

Kumar, Amit. “Rabin Karp Algorithm.” *TutorialCup*, 29 Dec. 2020, https://www.tutorialcup.com/interview/string/rabin-karp-algorithm.htm. Accessed 30 Sept. 2021.

Laud, Saharsh. “Knuth-Morris-Pratt Algorithm.” *CodesDope*, 24 Jan. 2021, https://www.codesdope.com/blog/article/kmp-algorithm/. Accessed 30 Sept. 2021.

"Prefix Function." CP-algorithms, cp-algorithms.com/string/prefix-function.html. Accessed 30 Sept. 2021.

Pöial, Jaanus. “String Algorithms - Exact Matching.” *Sõnetöötlus*, https://enos.itcollege.ee/~jpoial/algorithms/strings.html. Accessed 11 June 2021.

Popov, Stoimen. "Algorithm of the Week: Rabin-Karp String Searching." DZone, 3 Apr. 2012, dzone.com/articles/algorithm-week-rabin-karp. Accessed 11 June 2021.

“Rabin-Karp Algorithm.” *Brilliant Math & Science Wiki*, https://brilliant.org/wiki/rabin-karp-algorithm/. Accessed 11 June 2021.

“Rabin-Karp Algorithm for Pattern Searching.” *GeeksforGeeks*, 4 Sept. 2021, https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/. Accessed 11 June 2021.

“Rabin-Karp Algorithm.” *Programiz*, https://www.programiz.com/dsa/rabin-karp-algorithm. Accessed 11 June 2021.

“Rabin-Karp Algorithm | Searching for Patterns | Geeksforgeeks.” *YouTube*, YouTube, 12 June 2017, https://www.youtube.com/watch?v=oxd_Z1osgCk&feature=emb_title. Accessed 11 June 2021.

Rathod, Darshan. “Rabin-Karp Algorithm.” *Medium*, Medium, 10 Sept. 2019, https://medium.com/@darshanrathod4400/rabin-karp-algorithm-50bf47265b29. Accessed 11 June 2021.

“Rolling Hash Function Tutorial, Used By Rabin-Karp String Searching Algorithm.” *YouTube*, YouTube, 19 Dec. 2019, https://www.youtube.com/watch?v=BfUejqd07yo. Accessed 11 June 2021.

Soni, Kapil Kumar, et al. "Importance of String Matching in Real World Problems." Importance of String Matching in Real-World Problems, vol. 3, no. 6, 6 June 2014, ttp://www.ijecs.in/index.php/ijecs/article/view/651. Accessed 29 Sept. 2021.

Stephens, Poppy. “CS 5263 Bioinformatics Exact String Matching Algorithms. - Ppt Download.” *SlidePlayer*, https://slideplayer.com/slide/10737957/. Accessed 11 June 2021.

Techopedia. “What Is Hash Function? - Definition from Techopedia.” *Techopedia.com*, Techopedia, 26 Mar. 2018, https://www.techopedia.com/definition/19744/hash-function. Accessed 1 Oct. 2021.

“Time Complexity of Algorithms.” *Studytonight.com*, https://www.studytonight.com/data-structures/time-complexity-of-algorithms. Accessed 11 June 2021.

Budhwani, Girish. “String Matching (Kmp Algorithm).” *DEV Community*, DEV Community, 8 Nov. 2017, https://dev.to/girish3/string-matching-kmp-algorithm-cie. Accessed 11 June 2021.

Appendix

Section A: Programs Used

Knuth-Morris-Pratt algorithm (Geeksforgeeks) :

Code source: www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

```
# This code is contributed by Bhavya Jain
# Python program for KMP Algorithm
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    lps = [0]*M
    j = 0

    i = 0
    while i < N:
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == M:
            print ("Found pattern at index " + str(i-j))
            j = lps[j-1]

        elif i < N and pat[j] != txt[i]:
            if j != 0:
```

```
                    j = lps[j-1]
                else:
                    i += 1

def computeLPSArray(pat, M, lps):
    len = 0

    lps[0]
    i = 1

    while i < M:
        if pat[i]== pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            if len != 0:
                len = lps[len-1]
            else:
                lps[i] = 0
                i += 1

# Driver Code
txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
```

KMPSearch(pat, txt)

Rabin-Karp (geeksforgeeks) :

Code source: www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

 # This code is contributed by Bhavya Jain

```
d = 256

def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    i = 0
    j = 0
    p = 0
    t = 0
    h = 1
    for i in xrange(M-1):
        h = (h*d)%q

    for i in xrange(M):
        p = (d*p + ord(pat[i]))%q
        t = (d*t + ord(txt[i]))%q

    for i in xrange(N-M+1):
        if p==t:
            for j in xrange(M):
```

```
                if txt[i+j] != pat[j]:
                        break
                else: j+=1

            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j==M:
                print "Pattern found at index " + str(i)

        if i < N-M:
            t = (d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

            if t < 0:
                t = t+q

# Driver Code
txt = "GEEKS FOR GEEKS"
pat = "GEEK"

q = 101

search(pat,txt,q)
```

Section B: Full Results and Data Sets

Sample data:

Number: “7325679095170976”

(generated using https://www.random.org/strings/)

Dummy text: “Lorem ipsum dolor sit amet, consectetur adipiscing elit.”

(generated using https://www.blindtextgenerator.com/lorem-ipsum)

Random Strings: “owrdaotzbftsoczv”

(generated using https://www.random.org/strings/)

DNA Sequence: “GAGTGCCGCTTTCAGCCCCCCTGTCGTCGCCG”

(generated using http://www.faculty.ucr.edu/~mmaduro/random.htm)

Full results table (1-19):

Table 1:

Rabin Karp (numbers)	1	2	3	4	5	average time	Number of patterns
10	0.00004697	0.0000391	0.000036	0.00004292	0.00004792	0.000042582	1
100	0.00013113	0.0001359	0.0000701	0.00008798	0.00017691	0.000120404	1
1000	0.00036502	0.00039291	0.00034618	0.00037503	0.0028739	0.000870608	1
10000	0.00336599	0.00373101	0.00303507	0.00375795	0.00319409	0.003416822	1
100000	0.03619218	0.03725195	0.03696609	0.03820491	0.03570509	0.036864044	1
numbers (finding the middle value)							
KMP	1	2	3	4	5	average time	
10	0.00003791	0.00003505	0.00003982	0.00003314	0.00003505	0.000036194	1
100	0.00005603	0.00005794	0.00006008	0.00005698	0.00005603	0.000057412	1
1000	0.00024414	0.00023699	0.00026608	0.00026608	0.00025105	0.000252868	1
10000	0.00213099	0.00212002	0.00229192	0.00230098	0.00219989	0.00220876	1
100000	0.02290416	0.02999711	0.02640319	0.02488613	0.02405596	0.02564931	1

Table 2:

Rabin Karp (numbers)	1	2	3	4	5	average time	Number of patterns

10	0.00006294	0.00004601	0.00003815	0.00006008	0.00004101	0.000049638	1
100	0.00007892	0.00006795	0.0000689	0.00006819	0.00006795	0.000070382	1
1000	0.00063086	0.00037289	0.00100923	0.0007081	0.00094795	0.000733806	1
10000	0.00329804	0.00544	0.00366688	0.00412512	0.0034759	0.004001188	1
100000	0.03510904	0.04384899	0.03530407	0.03701782	0.08274698	0.04680538	1
numbers (finding the first value)							
KMP	1	2	3	4	5	average time	
10	0.000036	0.00004101	0.00004101	0.00005507	0.00004315	0.000043248	1
100	0.00005984	0.00005579	0.00007606	0.00009894	0.00006199	0.000070524	1
1000	0.00026989	0.00038218	0.00026703	0.00048208	0.00035906	0.000352048	1
10000	0.00379705	0.00223017	0.00342393	0.00216794	0.0030129	0.002926398	1
100000	0.02749181	0.02766418	0.05864501	0.03380609	0.02695203	0.034911824	1

Table 3:

Rabin Karp (numbers)	1	2	3	4	5	average time	Number of patterns
10	0.00007892	0.00004911	0.00005794	0.00026608	0.00004387	0.000099184	1
100	0.00006986	0.000067	0.000072	0.00006795	0.00007105	0.000069572	1
1000	0.00033498	0.00034118	0.00034094	0.00033689	0.00033307	0.000337412	1
10000	0.00589085	0.00621796	0.00550294	0.00308108	0.00311899	0.004762364	1
100000	0.03330588	0.03570914	0.03443885	0.03578782	0.04196405	0.036241148	
numbers (finding the last value)							
KMP	1	2	3	4	5	average time	
10	0.00003409	0.00006294	0.00004101	0.00005317	0.00006104	0.00005045	1
100	0.00005293	0.000067	0.00005603	0.00005913	0.00005388	0.000057794	1
1000	0.00055313	0.00026393	0.00030184	0.00023985	0.00023699	0.000319148	1
10000	0.00208998	0.00419688	0.00224018	0.00209594	0.00210404	0.002545404	1
100000	0.02265406	0.02201796	0.022753	0.04001188	0.02408719	0.026304818	1

Table 4:

Rabin Karp (DN sequence)	1	2	3	4	5	average time	Number of patterns
10	0.00004601	0.00010085	0.00010014	0.00036693	0.00024486	0.000171758	1
100	0.00019789	0.00016093	0.00007296	0.00007415	0.00011802	0.00012479	2
1000	0.00035191	0.00034404	0.001333	0.00078702	0.0004878	0.000660754	2
10000	0.00547099	0.004565	0.01099014	0.00470805	0.02046704	0.009240244	16
100000	0.1844368	0.05152488	0.04953599	0.16314507	0.23723912	0.137176372	109
DNA sequence finding the approximate middle value							
KMP	1	2	3	4	5	average time	
10	0.00017905	0.00007105	0.00003815	0.00005507	0.000072	0.000083064	1

100	0.00008416	0.00012612	0.00005984	0.00006294	0.00006509	0.00007963	2
1000	0.0004611	0.00025392	0.00052285	0.00026393	0.00058889	0.000418138	2
10000	0.00281215	0.00434995	0.00245595	0.00309706	0.00449705	0.003442432	16
100000	0.13638592	0.031178	0.10424709	0.06724691	0.17468905	0.102749394	109

Table 5:

Rabin Karp (DNA sequence)	1	2	3	4	5	average time	Number of patterns
10	0.00003815	0.000103	0.00005102	0.00008321	0.00004792	0.00006466	1
100	0.00006914	0.0000701	0.00009894	0.00023413	0.00012302	0.000119066	1
1000	0.00135899	0.0003891	0.00037289	0.00098109	0.00036502	0.000693418	1
10000	0.00817084	0.00601387	0.00651002	0.00826001	0.12026882	0.029844712	10
100000	0.05907011	0.12530303	0.22420287	0.30830002	0.14643693	0.172662592	109
DNA sequence finding the approximate first value							
KMP	1	2	3	4	5	average time	
10	0.00006795	0.00003409	0.00003481	0.00004792	0.00004196	0.000045346	1
100	0.00006008	0.0001061	0.00005794	0.00010586	0.00013399	0.000092794	1
1000	0.00103092	0.000283	0.00054502	0.00027704	0.00101304	0.000629804	1
10000	0.00310087	0.00250101	0.01319003	0.00239301	0.00435901	0.005108786	10
100000	0.03023291	0.34414196	0.03704691	0.03598499	0.03277588	0.09603653	109

Table 6

Rabin Karp (DNA sequence)	1	2	3	4	5	average time	Number of patterns
10	0.00013995	0.00003886	0.00007606	0.00005698	0.00003719	0.000069808	1
100	0.00007391	0.0000689	0.00008106	0.00012088	0.00007391	0.000083732	1
1000	0.00074601	0.00037909	0.000494	0.00038099	0.00057602	0.000515222	1
10000	0.00465202	0.00369692	0.01462793	0.04311705	0.01058316	0.015335416	9
100000	0.05075002	0.2193799	0.05955219	0.06270289	0.14460802	0.107398604	109
DNA sequence finding the approximate last value							
KMP	1	2	3	4	5	average time	
10	0.00003695	0.00008297	0.00003719	0.00003982	0.00006914	0.000053214	1
100	0.00005388	0.00015187	0.00005984	0.00005388	0.00005388	0.00007467	1
1000	0.00059915	0.0002811	0.00028205	0.00028515	0.00035596	0.000360682	1
10000	0.00386095	0.00831294	0.00573492	0.00496793	0.00472808	0.005520964	9
100000	0.03571606	0.17636704	0.03596497	0.07856917	0.02990317	0.071304082	109

Table 7:

Rabin Karp (randomly generated string)	1	2	3	4	5	average time	Number of patterns
10	0.00004482	0.00022984	0.00006795	0.00008297	0.00007296	0.000099708	1
100	0.00012803	0.00007105	0.0002079	0.00080419	0.00017405	0.000277044	1
1000	0.0003531	0.000494	0.00035214	0.00064206	0.00041008	0.000450276	1
10000	0.00498199	0.00585103	0.00336909	0.01315308	0.00431991	0.00633502	1
100000	0.05310488	0.05424619	0.03923917	0.05113816	0.05241609	0.050028898	1
string finding the approximate middle value							
KMP	1	2	3	4	5	average time	
10	0.00011587	0.000072	0.0001359	0.00009918	0.00006294	0.000097178	1
100	0.00019813	0.000067	0.00006413	0.00005889	0.00057983	0.000193596	1
1000	0.00023818	0.0002439	0.00043511	0.00050092	0.00023699	0.00033102	1
10000	0.00374317	0.00473094	0.00264287	0.00268197	0.00211596	0.003182982	1
100000	0.0242641	0.03579903	0.03150201	0.03402686	0.03015995	0.03115039	1

Table 8:

Rabin Karp (randomly generated string)	1	2	3	4	5	average time	Number of patterns
10	0.00009108	0.0000689	0.00007081	0.00004601	0.00007105	0.00006957	1
100	0.00012994	0.00086594	0.00009108	0.00024509	0.00014901	0.000296212	1
1000	0.00053811	0.00061512	0.00093985	0.00037789	0.00083399	0.000660992	1
10000	0.00387287	0.00504208	0.00356603	0.0030632	0.00457692	0.00402422	1
100000	0.06053901	0.06373501	0.04173398	0.03915215	0.04238486	0.049509002	1
string finding the approximate first value							
KMP	1	2	3	4	5	average time	
10	0.00005102	0.00010395	0.00007105	0.00006008	0.00007296	0.000071812	1
100	0.00011706	0.00012302	0.0000639	0.00011611	0.00009394	0.000102806	1
1000	0.00072098	0.00027704	0.00030613	0.00076509	0.00026894	0.000467636	1
10000	0.00475407	0.0024631	0.00211501	0.00558519	0.00368881	0.003721236	1
100000	0.0513401	0.03718615	0.02961302	0.0395689	0.02338386	0.036218406	1

Table 9:

Rabin Karp (randomly generated string)	1	2	3	4	5	average time	Number of patterns

10	0.00008416	0.00007296	0.00091505	0.00004601	0.000072	0.000238036	1
100	0.00014615	0.00014591	0.000875	0.00012088	0.00008893	0.000275374	1
1000	0.00033307	0.00096107	0.00035405	0.00040913	0.00037408	0.00048628	1
10000	0.00336003	0.00882006	0.00439715	0.00380087	0.00481296	0.005038214	1
100000	0.04394984	0.04490519	0.03600812	0.0537169	0.04171896	0.044059802	1
string finding the approximate last value							
KMP	1	2	3	4	5	average time	
10	0.00007296	0.00038505	0.00024605	0.0000689	0.00011206	0.000177004	1
100	0.00005984	0.00005507	0.00008893	0.00006986	0.00011992	0.000078724	1
1000	0.00047088	0.0002389	0.00051594	0.00050592	0.00049806	0.00044594	1
10000	0.00215077	0.00397778	0.00225115	0.00519395	0.00202799	0.003120328	1
100000	0.02126718	0.02360892	0.02593088	0.0245831	0.03724909	0.026527834	1

Table 10:

Rabin Karp (generated string)	1	2	3	4	5	average time	Number of patterns
10	0.00004196	0.00003791	0.00006795	0.00003815	0.00003791	0.000044776	1
100	0.00012207	0.00015998	0.00010109	0.00011611	0.00012302	0.000124454	1
1000	0.00049806	0.00045991	0.00036883	0.00147796	0.00038695	0.000638342	1
10000	0.00304198	0.00488305	0.00369596	0.00512695	0.00310707	0.003971002	1
100000	0.04383993	0.04586196	0.03758311	0.03918695	0.03952384	0.041199158	1
FINDING THE WORD (dummy text)							
KMP	1	2	3	4	5	average time	1
10	0.00005603	0.00004101	0.00006318	0.00004411	0.00004101	0.000049068	1
100	0.00008106	0.00008798	0.00010204	0.00009894	0.00011206	0.000096416	1
1000	0.00026107	0.00026488	0.00033712	0.00044179	0.00069094	0.00039916	1
10000	0.00238299	0.00311995	0.00269914	0.00206995	0.00323296	0.002700998	1
100000	0.03204584	0.03351617	0.02750206	0.02786994	0.0248971	0.029166222	1

Table 11:

Rabin Karp (numbers)	1	2	3	4	5	average time	Number of patterns
20	0.00319695	0.00548983	0.0040679	0.00366211	0.00448203	0.004179764	1
40	0.00311899	0.00323987	0.00338507	0.004076	0.00312591	0.003389168	1
60	0.0052042	0.00312901	0.00331593	0.00332093	0.00656104	0.004306222	1
80	0.00311208	0.00628901	0.00336409	0.00471616	0.00371599	0.004239466	1
100	0.00349522	0.004426	0.00411916	0.00313306	0.00310493	0.003655674	1
numbers (finding the middle value)							
KMP	1	2	3	4	5	average time	
20	0.00223899	0.00332403	0.004318	0.00271702	0.00299621	0.00311885	1

40	0.00243902	0.00379801	0.0036881	0.00261402	0.00217414	0.002942658	1
60	0.00262499	0.00353312	0.00379515	0.00218797	0.00235915	0.002900076	1
80	0.00342703	0.00277591	0.00277615	0.00323796	0.00243282	0.002929974	1
100	0.00221205	0.00225902	0.00220203	0.00271392	0.00360394	0.002598192	1

Table 12:

Rabin Karp (numbers)	1	2	3	4	5	average time	Number of patterns
20	0.00482798	0.00317097	0.00313497	0.00359297	0.00317192	0.003579762	1
40	0.00490284	0.00321817	0.00476694	0.00462914	0.00376987	0.004257392	1
60	0.00452805	0.00414705	0.00330687	0.0046401	0.00418997	0.004162408	1
80	0.004462	0.0032711	0.00496411	0.00337601	0.00339413	0.00389347	1
100	0.00384808	0.0032022	0.00463915	0.00376296	0.00314593	0.003719664	1
numbers (finding the first value)							
KMP	1	2	3	4	5	average time	
20	0.00217891	0.00429296	0.00319505	0.00259304	0.00218296	0.002888584	1
40	0.0032239	0.00228	0.0043571	0.002738	0.0032959	0.00317898	1
60	0.00407505	0.00218892	0.00431108	0.00241494	0.00405097	0.003408192	1
80	0.00298691	0.00301886	0.00229907	0.00419092	0.0021739	0.002933932	1
100	0.00306392	0.00228095	0.00327516	0.00374198	0.00333381	0.003139164	1

Table 13:

Rabin Karp (numbers)	1	2	3	4	5	average time	Number of patterns
20	0.00415802	0.00309014	0.00304103	0.00303817	0.00430989	0.00352745	1
40	0.00373602	0.00299215	0.00302196	0.00322986	0.00360894	0.003317786	1
60	0.00299406	0.00313997	0.00301385	0.00446081	0.00302386	0.00332651	1
80	0.00303888	0.00313997	0.00307512	0.0037539	0.00377417	0.003356408	1
100	0.00370598	0.00335717	0.0030849	0.00312185	0.004143	0.00348258	1
numbers (finding the last value)							
KMP	1	2	3	4	5	average time	
20	0.00277209	0.00217199	0.00212908	0.002177	0.00207305	0.002264642	1
40	0.00208116	0.00252891	0.00223613	0.00263	0.00223899	0.002343038	1
60	0.00235391	0.00243902	0.00209904	0.00346708	0.00304794	0.002681398	1
80	0.00210309	0.00213504	0.00212908	0.0022099	0.00240803	0.002197028	1
100	0.00208592	0.00237489	0.00320387	0.003757	0.00208592	0.00270152	1

Table 14:

Rabin Karp (DNA sequence)	1	2	3	4	5	average time	Number of patterns
20	0.00443101	0.00320888	0.00310206	0.00043392	0.003901	0.003015374	1
40	0.00308394	0.00310493	0.00344014	0.0030911	0.00310683	0.003165388	1
60	0.00316	0.00311899	0.005687	0.00317001	0.00355291	0.003737782	1
80	0.00355601	0.00323105	0.00440311	0.00328588	0.00392294	0.003679798	1
100	0.00323892	0.00317717	0.00342798	0.00320387	0.00459194	0.003527976	1
DNA sequence (finding the middle value)							
KMP	1	2	3	4	5	average time	
20	0.00227785	0.00244999	0.00343394	0.00326204	0.00334501	0.002953766	1
40	0.002671	0.00490594	0.00228	0.00288796	0.00237012	0.003023004	1
60	0.00282097	0.00243306	0.00289202	0.00371814	0.0053091	0.003434658	1
80	0.00346804	0.00226212	0.004287	0.00432086	0.0037601	0.003619624	1
100	0.00315905	0.00281596	0.00320005	0.00349307	0.00272012	0.00307765	1

Table 15:

Rabin Karp (DNA sequence)	1	2	3	4	5	average time	Number of patterns
20	0.00324678	0.00395393	0.0036869	0.0036869	0.00326514	0.00356793	1
40	0.00318503	0.00350499	0.00359201	0.00358915	0.00458097	0.00369043	1
60	0.00318289	0.00342298	0.00476813	0.00324798	0.00315499	0.003555394	1
80	0.00317788	0.00353813	0.00343895	0.00368595	0.00312304	0.00339279	1
100	0.00487304	0.00317001	0.00403881	0.00401497	0.00343204	0.003905774	1
DNA sequence (finding the first value)							
KMP	1	2	3	4	5	average time	
20	0.00339794	0.00231004	0.00233817	0.00230598	0.00443602	0.00295763	1
40	0.00352812	0.00352502	0.00338697	0.00353909	0.00233006	0.003261852	1considering increasing text and pattern sizes
60	0.00230384	0.00300097	0.00326896	0.00377393	0.002321	0.00293374	1
80	0.00367689	0.00229287	0.00385308	0.00229406	0.00241804	0.002906988	1
100	0.00379395	0.0026679	0.002599	0.00233603	0.00246787	0.00277295	1

Table 16:

Rabin Karp (DNA sequence)	1	2	3	4	5	average time	Number of patterns
20	0.00299215	0.00431204	0.00312901	0.00328708	0.00357795	0.003459646	1

40	0.00299501	0.00407314	0.00398302	0.00497508	0.00303507	0.003812264	1
60	0.00365782	0.00299716	0.00300598	0.00374413	0.0030179	0.003284598	1
80	0.00320601	0.00488114	0.00300908	0.00328803	0.00442314	0.00376148	1
100	0.00336218	0.00393891	0.00312996	0.00307488	0.00394607	0.0034904	1
DNA sequence (finding the last value)							
KMP	1	2	3	4	5	average time	
20	0.00240898	0.00234985	0.00228786	0.00228882	0.00325298	0.002517698	1
40	0.00350308	0.00320697	0.00223589	0.00219202	0.00223184	0.00267396	1
60	0.00283217	0.00219679	0.00219393	0.00224304	0.00219822	0.00233283	1
80	0.00238013	0.00253391	0.00225616	0.00352502	0.00246501	0.002632046	1
100	0.00225186	0.00234795	0.00222993	0.00231194	0.00279593	0.002387522	1

Table 17:

Rabin Karp (random string)	1	2	3	4	5	average time	Number of patterns
20	0.0030899	0.00380206	0.00307178	0.00308084	0.00324106	0.003257128	1
40	0.00314403	0.00311303	0.00358391	0.00308585	0.00311995	0.003209354	1
60	0.00327396	0.00330091	0.00368714	0.00424695	0.00432897	0.003767586	1
80	0.00312805	0.00405216	0.00309086	0.00362992	0.00308919	0.003398036	1
100	0.00311708	0.00502706	0.003093	0.00328302	0.00380611	0.003665254	1
random string (finding the middle value)							
KMP	1	2	3	4	5	average time	
20	0.00383115	0.00221181	0.00226712	0.00212598	0.00234199	0.00255561	1
40	0.00275016	0.00259709	0.00211191	0.0022428	0.00284386	0.002509164	1
60	0.00241184	0.00347996	0.00238299	0.00211501	0.00221205	0.00252037	1
80	0.00362682	0.00264502	0.00222301	0.00237393	0.00223899	0.002621554	1
100	0.00279808	0.00268602	0.00250602	0.00214791	0.00221682	0.00247097	1

Table 18:

Rabin Karp (random string)	1	2	3	4	5	average time	Number of patterns
20	0.00319195	0.00497103	0.0034461	0.0035789	0.003263	0.003690196	1
40	0.00310707	0.00310111	0.00320601	0.00343084	0.00328088	0.003225182	1
60	0.00315189	0.00317883	0.0033648	0.003299	0.00321102	0.003241108	1
80	0.00336409	0.00311613	0.00546408	0.00338602	0.00516081	0.004098226	1
100	0.003757	0.00317192	0.00447488	0.00327992	0.00317311	0.003571366	1
random string (finding the first value)							
KMP	1	2	3	4	5	average time	

20	0.00283504	0.0031271	0.00214601	0.00254822	0.00213218	0.00255771	1
40	0.00203681	0.00283504	0.00250816	0.00215697	0.00224113	0.002355622	1
60	0.00281	0.0023191	0.00319409	0.00216794	0.00357914	0.002814054	1
80	0.00310111	0.00239205	0.00252914	0.00290394	0.00213099	0.002611446	1
100	0.00215316	0.0030551	0.002491	0.00343299	0.00215411	0.002657272	1

Table 19:

Rabin Karp (random string)	1	2	3	4	5	average time	Number of patterns
20	0.00299191	0.00300097	0.00316191	0.00300789	0.00349617	0.00313177	1
40	0.00396299	0.00385189	0.00298309	0.00346303	0.00298405	0.00344901	1
60	0.00314689	0.00299406	0.00313807	0.00339508	0.00407696	0.003350212	1
80	0.00378203	0.00302601	0.00359511	0.00300694	0.00345802	0.003373622	1
100	0.00354695	0.0030179	0.00304198	0.00300288	0.00332999	0.00318794	1
random string (finding the last value)							
KMP	1	2	3	4	5	average time	
20	0.0020318	0.00334692	0.00205207	0.00311303	0.00264692	0.002638148	1
40	0.00203609	0.00328708	0.00237083	0.00205803	0.00211382	0.00237317	1
60	0.00248909	0.00228095	0.0020411	0.0030551	0.00222588	0.002418424	1
80	0.00203204	0.00203085	0.00237608	0.00203204	0.00217319	0.00212884	1
100	0.0022831	0.00204206	0.0022831	0.00305319	0.00210905	0.0023541	1